**✅ Modules Created**

**1. User Service**

* Created REST APIs for:
  + POST /api/users: Create a user
  + GET /api/users/{id}: Get a user by ID
* **Entity**: User
* **DTO**: UserDTO
* **Repository**: UserRepository
* **Service Layer**: Business logic
* **Controller**: REST endpoints

**🆕 New Concepts Used:**

* Spring Boot basic REST setup
* Spring Data JPA
* DTO usage to decouple entity from response

**2. Expense Service**

* Created REST APIs for:
  + POST /api/expenses: Create a new expense
  + GET /api/expenses/user/{userId}: Get all expenses for a user
* **Entity**: Expense (includes userId, amount, date, etc.)
* **Repository**: ExpenseRepository
* **Service Layer**:
  + Validates if user exists using UserService
  + Will check for total expenses vs budget
* **Controller**: Routes to service layer

**🆕 New Concepts Used:**

* Using LocalDate for date handling
* Service-to-service communication with Feign Client

**🔗 Microservice Communication**

**3. Feign Client**

* UserServiceClient created in Expense Service to call User Service:

java

CopyEdit

@FeignClient(name = "user-service", url = "http://localhost:8181/api/users")

public interface UserServiceClient {

@GetMapping("/{id}")

User getUserById(@PathVariable Long id);

}

**🆕 New Concepts Used:**

* Spring Cloud OpenFeign
* Declarative REST client
* Service integration with external APIs

**💡 Business Logic**

**4. User Validation**

* When adding an expense, user is first validated via Feign Client.
* If not found, throw:

java

CopyEdit

throw new ResponseStatusException(HttpStatus.BAD\_REQUEST, "User not found in User Service");

**To be improved** (✅ next up): Move to **Custom Exception + Global Exception Handler**

**⚠️ Planned Business Logic (Not Fully Implemented Yet)**

**5. Budget Validation Feature (Planned)**

* Before creating a new expense:
  + Get total expense amount of the user (from DB)
  + Compare with user's budget (to be added to User Entity)
  + If total + new > budget → **throw budget exceeded error**

**🆕 Will require:**

* Adding budget field to User entity
* Aggregation logic (sum(expenses.amount)) from ExpenseRepository

**⚙️ Setup & Configuration**

**6. Spring Boot + Spring Cloud Compatibility Fix**

* Issue: Spring Boot 3.5.3 not compatible with Spring Cloud
* Solution: Downgraded to **Spring Boot 3.3.x** or changed Spring Cloud version

**🧪 Testing**

**7. Tested with Postman**

* Manually verified:
  + Expense creation
  + Error response when user is not found
  + Data saved in MySQL
* Planned: Add more test scenarios for budget alerts, validations, and aggregations

**🧱 Dependencies Used**

* Spring Boot Starter Web
* Spring Boot Starter Data JPA
* MySQL Connector
* Lombok
* Spring Cloud OpenFeign
* DevTools (for live reload)

**📌 Next Immediate Tasks**

* ✅ Add budget field to User entity
* ✅ Calculate total expenses for a user in ExpenseService
* ✅ Validate budget before allowing expense creation
* ❌ Replace ResponseStatusException with a **custom exception**
* ❌ Implement @RestControllerAdvice with a global exception handler

**🔄 Comparison Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Type | Pros | Cons |
| Feign | Declarative | Clean, easy, Spring Cloud native | Blocking, not reactive |
| RestTemplate | Imperative | Simple for small use cases | Deprecated, verbose, blocking |
| WebClient | Reactive | Non-blocking, modern, more powerful | Slightly more complex |
| gRPC | Binary | Very fast, schema-based | More setup, not HTTP/JSON |
| Kafka/Rabbit | Async | Decoupled, resilient | Complex, harder to trace failures |

**📌 Try and catch to validate the issue**

public void validateUserExists(Long userId) {  
 try {  
 userClient.getUserById(userId); // Feign client call  
 } catch (FeignException.NotFound ex) {  
 throw new ResponseStatusException(HttpStatus.BAD\_REQUEST, "User not found in User Service");  
 } catch (FeignException ex) {  
 throw new ResponseStatusException(HttpStatus.SERVICE\_UNAVAILABLE, "User Service error");  
 }  
}

**✅ USE CASE: validateUserExists(Long userId) Custom Error**

**🔄 Step-by-Step Flow**

**1. 🚀 A request comes in**

You make a POST request like:

json

CopyEdit

{

"title": "Groceries",

"amount": 5000,

"userId": 1

}

to the ExpenseController.createExpense(...) endpoint.

**2. 🧠 Controller calls the Service Layer**

java

CopyEdit

expenseService.createExpense(expenseDto);

Inside createExpense(), we first **validate if the user exists**:

java

CopyEdit

validateUserExists(expenseDto.getUserId());

**3. 📞 Feign Client makes HTTP call**

The Feign client (userClient.getUserById(userId)) sends a REST API request to the **User Service**.

If userId 1 exists → proceeds normally.

If userId 1 does **not exist** → User Service returns **404 Not Found**.

**4. ⚠️ FeignException is thrown**

If user is not found, Feign throws a FeignException.NotFound.

This is caught in your code:

java

CopyEdit

catch (FeignException.NotFound ex) {

throw new UserNotFoundException("User with ID " + userId + " not found in User Service");

}

Now your custom exception UserNotFoundException is thrown.

**5. 🎯 Global Exception Handler catches it**

Spring finds a matching @ExceptionHandler in your @ControllerAdvice class:

java

CopyEdit

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<?> handleUserNotFound(UserNotFoundException ex)

**6. 🧾 Returns Custom Error Response**

You return a structured JSON response like this:

json

CopyEdit

{

"timestamp": "2025-07-09T23:55:12",

"status": 400,

"error": "Bad Request",

"message": "User with ID 1 not found in User Service"

}

**📌 Summary of Flow (Diagram Style):**

css

CopyEdit

[POST Request] ──▶ [Controller]

│

▼

[ExpenseService.createExpense()]

│

▼

[validateUserExists(userId)]

│

[Feign Client calls User Service]

│

▼

User Not Found → throws FeignException.NotFound

│

▼

[throw new UserNotFoundException()]

│

▼

[GlobalExceptionHandler catches it via @ControllerAdvice]

│

▼

[Returns clean JSON error to Postman/client]

**✅ Benefits of This Architecture:**

| **Feature** | **Benefit** |
| --- | --- |
| 🔍 Clear Exceptions | Custom messages per case |
| 🧼 Clean Code | No ugly if-else checks in controller |
| 🔁 Reusability | Same handler can be used across services |
| 🧪 Easy Testing | You can unit test exception scenarios easily |
| 🔧 Extendable | Add more exception types like BudgetLimitExceededException, etc. |

Great question! Let's break down **how the @FeignClient works**, and **how it forms the URL** behind the scenes when you just call userClient.getUserById(userId).

**🔧 Your Feign Client Setup:**

java

CopyEdit

@FeignClient(name = "user-service", url = "http://localhost:8181")

public interface UserClient {

@GetMapping("/api/users/{id}")

UserDto getUserById(@PathVariable("id") Long userId);

}

**🔄 How the Feign Client Forms the URL and Makes the Call**

When you call this method:

java

CopyEdit

userClient.getUserById(1L);

Here's what happens step-by-step:

**1. 🎯 Base URL comes from url = "http://localhost:8181"**

That’s the server address where your **User Service** is running.

**2. 🧩 Path is defined by @GetMapping("/api/users/{id}")**

So Feign knows it should hit this endpoint:

bash

CopyEdit

GET /api/users/1

**3. 🧠 Path Variable is replaced**

{id} is replaced by the userId you passed in:

java

CopyEdit

@FeignClient(...)

@GetMapping("/api/users/{id}")

UserDto getUserById(@PathVariable("id") Long userId);

If userId = 1, final URL becomes:

bash

CopyEdit

http://localhost:8181/api/users/1

**✅ So this simple line:**

java

CopyEdit

userClient.getUserById(1L);

**Under the hood becomes:**

bash

CopyEdit

HTTP GET http://localhost:8181/api/users/1

**🔥 Feign Handles All Of This For You:**

* URL building
* HTTP request
* JSON deserialization into UserDto
* Throwing exceptions (FeignException) if status ≠ 200

**💡 Why It’s Powerful:**

You don’t have to write:

java

CopyEdit

RestTemplate rest = new RestTemplate();

UserDto user = rest.getForObject("http://localhost:8181/api/users/1", UserDto.class);

Feign does all that with just an interface call.

Yes — if you're writing **custom exceptions** in a Spring Boot project, then creating a **global exception handler** is not only recommended but **necessary** if you want:

* **Centralized error handling** across your whole application
* **Consistent error responses** (like message, timestamp, status)
* To catch your **custom exceptions** and return proper HTTP responses (instead of 500 Internal Server Errors)

**✅ Here's What You Need:**

**1. Create a Custom Exception (e.g., UserNotFoundException)**

java

CopyEdit

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {

super(message);

}

}

**2. Create a Global Exception Handler**

java

CopyEdit

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<Map<String, Object>> handleUserNotFound(UserNotFoundException ex) {

Map<String, Object> error = new HashMap<>();

error.put("timestamp", LocalDateTime.now());

error.put("status", HttpStatus.NOT\_FOUND.value());

error.put("error", "User Not Found");

error.put("message", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class) // fallback for all other exceptions

public ResponseEntity<Map<String, Object>> handleGeneric(Exception ex) {

Map<String, Object> error = new HashMap<>();

error.put("timestamp", LocalDateTime.now());

error.put("status", HttpStatus.INTERNAL\_SERVER\_ERROR.value());

error.put("error", "Internal Server Error");

error.put("message", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**⚙️ How It Works:**

* When your service throws throw new UserNotFoundException("User not found in User Service")
* Spring checks for a matching @ExceptionHandler in @RestControllerAdvice
* Returns a JSON error response with the exact structure you want

**📦 Summary:**

| **Component** | **Purpose** |
| --- | --- |
| CustomException | Define specific errors like UserNotFound |
| @RestControllerAdvice | Handle all exceptions globally |
| @ExceptionHandler | Maps specific exceptions to HTTP responses |

com/

└── expensetracker/

├── expense\_service/

├── user\_service/

└── exception/

├── UserNotFoundException.java

├── BudgetExceededException.java

└── GlobalExceptionHandler.java

**ChatGPT said:**

Great question! You’re right — **you *can*** directly throw an error using ResponseStatusException like:

java

CopyEdit

throw new ResponseStatusException(HttpStatus.BAD\_REQUEST, "User not found in User Service");

This works perfectly for **simple one-off use cases**.

**🔍 But here’s *why* and *when* we use custom exceptions:**

**✅ 1. Clean Separation of Concerns**

Instead of hardcoding HTTP logic (ResponseStatusException) inside your **service layer**, you just throw:

java

CopyEdit

throw new UserNotFoundException("User not found in User Service");

This separates **business logic** from **HTTP response handling**, which is cleaner and easier to manage.

**✅ 2. Reusability**

You may need the same exception in multiple places:

java

CopyEdit

throw new UserNotFoundException("User not found in DB");

Instead of repeating new ResponseStatusException(...) everywhere, you reuse your UserNotFoundException.

**✅ 3. Centralized Error Handling (@RestControllerAdvice)**

Using custom exceptions lets you return **custom error formats**, log errors, and set consistent HTTP status codes *automatically*:

java

CopyEdit

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<ErrorResponse> handleUserNotFound(UserNotFoundException ex, HttpServletRequest request) {

return new ResponseEntity<>(

new ErrorResponse(400, ex.getMessage(), request.getRequestURI()),

HttpStatus.BAD\_REQUEST

);

}

You write it once → works for every controller.

**✅ 4. Makes Testing Easier**

When writing unit tests, you can assert on specific exception types like:

java

CopyEdit

assertThrows(UserNotFoundException.class, () -> service.getUser(5L));

You can't do this reliably with generic ResponseStatusException.

**🚫 Downsides of only using ResponseStatusException:**

* Harder to maintain when logic grows.
* Repeated HTTP-related code in services.
* Inconsistent error structure unless handled globally.

**✅ So the ideal approach:**

* Throw **custom exceptions** from services.
* Let **@RestControllerAdvice** decide what response should go out.
* Keep **service layer focused only on logic**.

**Summary:**

|  |  |  |
| --- | --- | --- |
| Scenario | Use ResponseStatusException | Use Custom Exception |
| Quick one-liner logic | ✅ Yes | ❌ Not needed |
| Repeated logic across classes | ❌ No | ✅ Yes |
| You want consistent error format | ❌ No | ✅ Yes |
| You want global error handling | ❌ Hacky | ✅ Clean |
| Easy unit testing | ❌ No | ✅ Yes |

Let me know if you want a sample structure to keep your exceptions organized.

**💥 1. Exception is Thrown Inside ExpenseService.java:**

java

CopyEdit

catch (FeignException.NotFound ex) {

throw new UserNotFoundException("User not found in User Service");

}

👉 Here, you're **catching** the FeignException.NotFound (which is an external/internal service exception)  
👉 But instead of throwing that **third-party exception**, you throw your own custom, **clean exception**:  
UserNotFoundException.

This makes your **codebase cleaner**, hides internal implementation details, and helps return consistent error responses.

**📦 2. Custom Exception UserNotFoundException is Thrown**

Your exception class:

java

CopyEdit

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {

super(message);

}

}

Even though we only pass a **message**, this is enough to **trigger Spring's exception mechanism**, because it **extends RuntimeException**.

**🛡 3. @ControllerAdvice + @ExceptionHandler Catches It**

In your GlobalExceptionHandler.java, you have something like:

java

CopyEdit

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<ErrorResponse> handleUserNotFound(UserNotFoundException ex) {

ErrorResponse error = new ErrorResponse();

error.setMessage(ex.getMessage());

error.setStatus(HttpStatus.NOT\_FOUND.value());

error.setTimeStamp(System.currentTimeMillis());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

}

✅ **Even though you didn’t pass the original exception (ex)**, the custom UserNotFoundException is still caught by this handler.  
Why?

Because:

* The **type matches** (@ExceptionHandler(UserNotFoundException.class))
* The **message** is still carried (ex.getMessage())
* Spring Boot **knows which handler to route it to**

**✅ 2. Budget Validation Feature**

**📍 In ExpenseService.java**

* Added method:

java

CopyEdit

public void validateUserAndBudget(Long userId, BigDecimal newExpenseAmount)

* + Fetches user via Feign client (UserClient)
  + Calculates total expenses for that user (no monthly check)
  + Compares total + new expense with monthlyBudget
  + Throws BudgetExceededException if budget exceeded
* Simplified by removing **month/year filter** for now

**✅ 3. User Integration via Feign**

* Added UserDto.java to map user details

java

CopyEdit

private Long id;

private String name;

private BigDecimal monthlyBudget;

* Created Feign client UserClient:

java

CopyEdit

@FeignClient(name = "user-service", url = "http://localhost:8181")

public interface UserClient {

@GetMapping("/api/users/{id}")

UserDto getUserById(@PathVariable("id") Long userId);

}

**✅ 4. Repository Update**

* Added repository method:

java

CopyEdit

@Query("SELECT COALESCE(SUM(e.amount), 0) FROM Expense e WHERE e.userId = :userId")

BigDecimal getTotalSpentByUser(@Param("userId") Long userId);

* Removed previous monthly summary query and its endpoint

**🔒 How security typically works in microservices**

You have 2 options:

1. **Each service has its own Spring Security + JWT validation**
   * Every service checks the JWT itself.
   * The token is issued by user\_service, and others just verify it.
   * Simple, no extra infra.
2. **Use an API Gateway (recommended for bigger setups)**
   * Only the gateway validates JWT.
   * Internal services (user, expense, etc.) trust the gateway and don’t handle security individually.
   * Cleaner, but requires setting up a gateway (e.g. Spring Cloud Gateway).

\*\*JWT Authentication in Spring Boot: Complete Notes\*\*

---

\*\*Why JWT?\*\*

- JWT (JSON Web Token) enables stateless, secure authentication.

- No server-side session storage; token contains user info and is verified on each request.

---

\*\*Key Components & Their Roles\*\*

1. \*\*AuthController\*\*

- Handles login API (`/auth/login`).

- Receives credentials, calls authentication service.

- Returns JWT token if authentication succeeds.

2. \*\*DTOs (Data Transfer Objects)\*\*

- Used for request/response payloads (e.g., `LoginRequest`, `LoginResponse`).

3. \*\*CustomUserDetailsService\*\*

- Loads user details from the database.

- Integrates with Spring Security for authentication.

4. \*\*JwtUtil\*\*

- Generates JWT tokens (`generateToken`).

- Extracts username from token (`extractUsername`).

- Validates token (`validateToken`).

5. \*\*JwtAuthenticationFilter\*\*

- Intercepts every request.

- Extracts JWT from `Authorization` header.

- Validates token and sets authentication in `SecurityContextHolder`.

6. \*\*SecurityConfig\*\*

- Configures Spring Security.

- Registers `JwtAuthenticationFilter`.

- Secures endpoints, sets up authentication manager.

---

\*\*API Call Flow\*\*

1. \*\*Client sends login request\*\* to `/auth/login` with credentials.

2. \*\*AuthController\*\* receives request, calls authentication service.

3. \*\*Service\*\* uses `CustomUserDetailsService` to load user and checks password.

4. If valid, \*\*JwtUtil\*\* generates JWT token.

5. \*\*AuthController\*\* returns JWT token to client.

6. \*\*Client stores JWT\*\* (e.g., local storage).

7. For protected APIs, \*\*client sends JWT\*\* in `Authorization: Bearer <token>` header.

8. \*\*JwtAuthenticationFilter\*\* intercepts request, validates token using `JwtUtil`.

9. If valid, \*\*SecurityContextHolder\*\* is set with user details.

10. \*\*Controller/service\*\* methods can access authenticated user info.

---

\*\*Summary Table\*\*

| Component | Purpose/Task |

|--------------------------|---------------------------------------------------|

| AuthController | Handles login, returns JWT |

| DTOs | Structures request/response data |

| CustomUserDetailsService | Loads user from DB for authentication |

| JwtUtil | Creates, validates, parses JWT |

| JwtAuthenticationFilter | Secures endpoints, validates JWT on each request |

| SecurityConfig | Configures security, registers JWT filter |

---

\*\*Why Each Step?\*\*

- \*\*JWT\*\*: Stateless, scalable authentication.

- \*\*Controller/DTOs\*\*: Clean API structure.

- \*\*UserDetailsService\*\*: DB integration for user info.

- \*\*JwtUtil\*\*: Central JWT logic.

- \*\*Filter\*\*: Secures endpoints.

- \*\*SecurityConfig\*\*: Spring Security setup.

---

\*\*Flow Recap:\*\*

Login API → AuthController → Service → UserDetailsService → JwtUtil → Return JWT → Client stores JWT → Sends JWT in header → Filter validates → SecurityContext set → Access protected resources.

---

\*\*Use these notes for revision and understanding the JWT authentication flow in Spring Boot.\*\*

**🔑 JWT Token Flow (Spring Boot + Postman)**

**1. Login & Token Generation**

* User calls **/auth/login** with username & password.
* If credentials are valid:
  + JwtUtil generates a JWT token.
  + Response contains the token:
  + { "token": "eyJhbGciOi..." }

**2. Using the Token**

* For every protected API request:
  + Add header →
  + Authorization: Bearer <your-token>
* Example (Postman):
  + Go to **Authorization tab → Bearer Token** → Paste the token.
  + Or manually in Headers:
  + Key: Authorization
  + Value: Bearer eyJhbGciOi...

**3. Validation Process**

* Every incoming request passes through JwtAuthenticationFilter.
* The filter:
  1. Extracts token from **Authorization header**.
  2. Validates the token using JwtUtil.
  3. If valid → sets authentication in SecurityContext.
  4. If invalid/missing → returns 401 Unauthorized.

**4. Common Errors**

* **401 Unauthorized** → Token invalid/expired/not sent.
* **403 Forbidden** → Token valid but access to the endpoint is not allowed by SecurityConfig.
* Wrong header key (like token instead of Authorization).

**5. Best Practices**

* Keep login endpoint (/auth/login) **public**.
* Protect all other APIs via JWT.
* Use **Postman Authorization tab** for easier testing.
* (Later) Move JWT validation to **API Gateway** so microservices don’t repeat the same logic.

👉 This is the **core cycle of JWT**:  
Login → Get Token → Send Token in Header → Filter validates → Access granted or denied.

Great question Niraj 👍 This is the perfect time to understand **Eureka Service Registry** and **API Gateway** before we add them to your project.

**⚡ 1. Eureka Service Registry**

**🔹 What it is:**

* Eureka is a **service registry** from Spring Cloud Netflix.
* Think of it like a **“yellow pages”** for your microservices: every service (User Service, Expense Service, etc.) **registers itself** with Eureka when it starts.
* Other services (or the gateway) can **discover services dynamically** instead of using hardcoded URLs.

**🔹 Why we need it:**

* Right now, in your Expense Service Feign Client, you’re calling http://localhost:8181/api/users.
* But what if tomorrow:
  + The User Service runs on a different port?
  + You deploy multiple instances for load balancing?
* With Eureka:
  + You just call lb://user-service (logical name).
  + Eureka resolves it to the correct running instance automatically.

**🔹 How we’ll use it:**

* Create a new project discovery-server (Eureka server).
* Run it on port 8761.
* Add both User Service & Expense Service as **Eureka clients** → they auto-register.
* Change Feign Client to use service name (@FeignClient(name="user-service")) instead of hardcoded localhost.

**⚡ 2. API Gateway (Spring Cloud Gateway)**

**🔹 What it is:**

* API Gateway is the **single entry point** for all external requests.
* Clients don’t directly call User Service or Expense Service anymore.
* Instead, all requests go through the gateway, which:
  + Routes them to the correct microservice.
  + Can apply cross-cutting concerns: **authentication, logging, rate limiting, etc.**

**🔹 Why we need it:**

* Currently, your frontend (or Postman) must know two URLs:
  + http://localhost:8181/api/users
  + http://localhost:8282/api/expenses
* That’s messy and tightly coupled.
* With Gateway:
  + Client just calls http://localhost:8080/api/users/... or http://localhost:8080/api/expenses/...
  + Gateway routes the request to the right service automatically.
* Security (JWT validation) can happen **once at the gateway** → services don’t need to repeat the same filter.

**🔹 How we’ll use it:**

* Create a new project api-gateway.
* Register it with Eureka.
* Define routes in application.yml:
* spring:
* cloud:
* gateway:
* routes:
* - id: user-service
* uri: lb://user-service
* predicates:
* - Path=/api/users/\*\*
* - id: expense-service
* uri: lb://expense-service
* predicates:
* - Path=/api/expenses/\*\*
* Add a **JWT filter** in the gateway → validates token.
* If valid → forwards request to the microservice.
* If invalid → blocks it at the gateway.

**🎯 Summary**

* **Eureka** → Service registry (dynamic discovery, no hardcoded URLs).
* **API Gateway** → Single entry point (routing + security + cross-cutting features).
* **Why in our project?**
  + Makes microservices scalable, flexible, and cleaner.
  + Frontend talks to only one URL.
  + JWT validation centralized at the gateway.

so technically gate way is a service will will call other service ?

**ChatGPT said:**

Exactly Niraj ✅ You got it!

Think of the **API Gateway** as just **another Spring Boot service**, but with a **special role**:

**🔑 How Gateway Works (in simple terms)**

1. **Client → Gateway**
   * The client (Postman / frontend / mobile app) sends every request **only to the Gateway**.
   * Example:
   * http://localhost:8080/api/users/1
   * http://localhost:8080/api/expenses
2. **Gateway → Other Services**
   * The Gateway looks at the path (/api/users/... or /api/expenses/...)
   * Then it **forwards the request** to the correct service (User Service, Expense Service).
   * It’s basically a smart **reverse proxy**.
3. **Add Extra Features at Gateway**
   * Since every request passes through the Gateway, we can add **common logic once**:
     + ✅ JWT validation (auth check)
     + ✅ Logging / monitoring
     + ✅ Rate limiting
     + ✅ Load balancing (if multiple instances of a service exist)
4. **Response Back to Client**
   * Gateway takes the response from the microservice and passes it back to the client.

**📌 Analogy**

* Without Gateway → You directly call each shop (User Service, Expense Service). You need to know all their addresses.
* With Gateway → You go to a **shopping mall entrance (Gateway)**. Tell it “I want groceries (Expense Service)” → It forwards you to the right shop.

**🎯 Why this is important for your project**

* Today you have just **2 services** (User + Expense).
* Tomorrow you might add: Notification Service, Budget Service, Reporting Service…
* Instead of exposing all these services to the client, you expose only **one gateway**.
* Easier for frontend, more secure, and scalable.

there are two service serice and experse trcaker service where to creat thie and why

**ChatGPT said:**

Good catch Niraj 👍 Let’s clarify where the **Eureka Server** fits in your setup.

**📌 Current Situation (your project)**

* You already have **two microservices**:
  1. **User Service** (handles users + JWT login).
  2. **Expense Service** (handles expenses, calls User Service with Feign).

**📌 Where to create Eureka (Discovery Server)?**

* Eureka is **not part of User Service or Expense Service**.
* It’s a **new, separate Spring Boot project** that acts as the **Service Registry**.
* Think of it like a **central directory**:
  + User Service registers itself.
  + Expense Service registers itself.
  + Later, API Gateway will also register.

📍 So, you create a **new project folder** → discovery-server/ (like I gave you above).

**📌 Why separate?**

* Microservices = **loosely coupled** → each has its own job.
* User Service → users.
* Expense Service → expenses.
* Discovery Server → service registry (manages addresses of other services).
* If we mixed them, you’d lose the benefit of **scalability** and **independent deployment**.

Got it 👍 Here’s a clean summary/notes of what you’ve done for **User-Service registration with Eureka** — you can use this later to revise 👇

**📘 Notes: Registering User-Service with Eureka**

**🔹 What is Eureka?**

* **Eureka** is a **Service Registry** used in microservices architecture.
* All services (User, Expense, etc.) register themselves here.
* Other services and API Gateway can discover them **without hardcoding IP/port**.

**🔹 Steps to Register User-Service**

**1. Add Dependency in pom.xml**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

**2. Enable Eureka Client in main class**

UserServiceApplication.java:

@SpringBootApplication

@EnableDiscoveryClient

public class UserServiceApplication {

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

}

**3. Configure application.properties**

spring.application.name=user-service

server.port=8081

# Register with Eureka

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

**4. Run and Verify**

* Start **discovery-server**.
* Start **user-service**.
* Visit: http://localhost:8761
* ✅ You should see **user-service** under *Instances currently registered with Eureka*.

**🔹 Why is this important?**

* Allows **dynamic service discovery** (no hardcoding service URLs).
* Makes microservices **scalable** and **fault-tolerant**.
* API Gateway will later use Eureka to find and route requests.

**Notes: Feign Client JWT Token Propagation - Issue, Fix & Flow**

**What We Did Today**

* Implemented Feign Client in Expense Service to call User Service.
* Tried to get JWT token from Spring Security context in Expense Service.
* Added Spring Security dependency but did not configure it.
* Faced 401 Unauthorized and CSRF errors due to missing token in Feign calls.
* Realized Expense Service does not have Spring Security or authentication configured.
* Changed Feign interceptor to read the Authorization header directly from incoming HTTP requests.
* Removed dependence on Spring Security context in Expense Service.
* Clients (frontend/Postman) must send JWT token in Authorization header to Expense Service.
* Expense Service forwards the token through Feign calls transparently.
* User Service validates JWT token and responds accordingly.

**Problem Summary**

* Expense Service’s SecurityContext is empty because it does not perform authentication.
* Trying to get JWT token from SecurityContext fails.
* Feign calls miss Authorization header → User Service rejects with 401.
* Adding security dependency without config causes blocking filters and CSRF errors.

**Solution Summary**

* Use Spring’s RequestContextHolder in Feign interceptor to get raw HTTP Authorization header.
* Forward this header as-is in Feign client requests to User Service.
* Expense Service remains unsecured, acting as a relay.
* User Service remains the sole JWT validator.

Token and Call Flow Diagram

text

Client

└──(JWT Token in Authorization header)──▶ Expense Service (no auth)

└──(Feign Call - Authorization header relayed)──▶ User Service (auth + JWT validation)

└──── Validates JWT, responds ──────▶ Expense Service ───▶ Client

| **Aspect** | **Before Fix** | **After Fix** |
| --- | --- | --- |
| Expense Service Security | Dependency added but no config; CSRF errors | No security dependency, no config needed |
| Token Retrieval Method | From Spring Security context (empty) | From HTTP request header (Authorization) |
| Feign Client Header | Missing Authorization header | Authorization header forwarded correctly |
| User Service Response | 401 Unauthorized | Valid response |

**What is API Gateway?**

* API Gateway is the **single entry point for all client requests**.
* It acts as a **reverse proxy**, routing requests to appropriate microservices.
* It manages **cross-cutting concerns** such as:
  + **Authentication and authorization** (validate tokens once here)
  + **Logging and monitoring**
  + **Rate limiting and throttling**
  + **Load balancing**
* Simplifies client interaction by exposing a **unified API** rather than many microservice endpoints.

**Why Use an API Gateway?**

* Avoids clients calling each microservice directly — centralizes traffic.
* Enables **decoupling** between frontend and microservices.
* Handles **security** consistently in one place.
* Supports scaling services independently.
* Easier to add common features (e.g., CORS, metrics) across all API calls.

**How to Set Up API Gateway Using Spring Cloud Gateway**

**1. Create a new Spring Boot project for the gateway**

* Select dependencies:
  + **Spring Cloud Gateway**
  + **Eureka Client**

**2. Add dependencies (pom.xml)**

xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

**3. Enable Eureka Client**

java

@SpringBootApplication

@EnableEurekaClient

**public** **class** ApiGatewayApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(ApiGatewayApplication.**class**, args);

}

}

**4. Configure application.properties (or .yml)**

text

server.port=8080

spring.application.name=api-gateway

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

**5. Define routes by service name and path**

Example in application.yml or properties:

text

spring:

cloud:

gateway:

routes:

- id: user-service

uri: lb://user-service

predicates:

- Path=/api/users/\*\*

- id: expense-service

uri: lb://expense-service

predicates:

- Path=/api/expenses/\*\*

This routes all /api/users/\*\* to User Service, /api/expenses/\*\* to Expense Service.

**6. (Optional) Add JWT authentication filter at gateway**

* Centralize token validation.
* Forward authenticated requests downstream.

**What You Have Now**

* Your pom.xml is mostly correct, with spring-cloud-starter-netflix-eureka-client dependency.
* In your main Spring Boot application class, you have @EnableEurekaClient but IntelliJ shows it as red (unrecognized).
* You want to know what to change to fix this and have Eureka client properly register.

**What to Change and Why**

1. **Remove or Ignore the @EnableEurekaClient Annotation**

* Recent Spring Cloud versions (including your version 2025.0.0) **auto-configure Eureka client** if the Eureka client dependency is present.
* You **do not need** to annotate your main application class with @EnableEurekaClient anymore.
* Simply use the annotation-less main class:

Yes, you added the Swagger dependency in the **Gateway** too — that’s why you expected http://localhost:8080/swagger-ui.html to work.

But here’s the key point 👇

🔹 **Swagger/OpenAPI (springdoc) generates docs only for the APIs available inside that application.**

* Your **Gateway** doesn’t have any REST controllers of its own (it just routes to Expense & User services).
* So Swagger in the Gateway has **nothing to document**, which is why you’re getting **404**.

**✅ Two options for you**

**Option 1: Swagger per service (easier)**

* Keep Swagger in **User-Service (8181)** and **Expense-Service (8282)** only.
* Access them directly:
* http://localhost:8181/swagger-ui/index.html
* http://localhost:8282/swagger-ui/index.html

**Option 2: Swagger Aggregation in API Gateway (advanced)**

If you want **one Swagger UI in the Gateway** that shows **both services’ APIs**, you need **Swagger aggregation**.

For that, add this dependency in **Gateway**:

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-api</artifactId>

<version>2.5.0</version>

</dependency>

Then configure **Springdoc Aggregator** in application.yml (inside Gateway):

springdoc:

swagger-ui:

urls:

- name: Expense Service

url: http://localhost:8282/v3/api-docs

- name: User Service

url: http://localhost:8181/v3/api-docs

Now, when you go to:

http://localhost:8080/swagger-ui/index.html

you’ll see **both services’ APIs** in one UI.

👉 Do you want me to write the **full Gateway application.yml config** for aggregation (with Eureka service names instead of localhost), so you don’t need to hardcode ports?

**🔑 Summary**

* **Problem**: Swagger docs were blocked by security.
* **Fix**: Allowed Swagger endpoints in SecurityConfig.
* **Gateway setup**: Added Swagger aggregation dependency + URLs in application.properties.
* **Result**: Unified Swagger UI in Gateway + Secured APIs with JWT.

**Security Config update**

* Allowed **Swagger endpoints** (/v3/api-docs/\*\*, /swagger-ui/\*\*, /swagger-ui.html).
* Kept login (/auth/\*\*) and signup (POST /api/users) public.
* Protected /api/admin/\*\* (ADMIN only) and /api/user/\*\* (USER + ADMIN).
* Everything else requires JWT authentication.

✅ This fixed the issue where Swagger UI wasn’t loading because it was being blocked by Spring Security.